Parte QUATRO

Streams e Coleções

Capítulo DOZE – Streams

Objetivos do Exame

- Descrever a interface Stream e o pipeline de Stream.
- Usar referências de métodos com Streams.

Um exemplo simples

Suponha que você tenha uma lista de estudantes e os requisitos sejam extrair os estudantes com uma nota de **90.0 ou superior** e classificá-los por nota em ordem crescente.

Uma maneira de fazer isso seria:

java Copiar Editar

```java
List<Student> studentsScore = new ArrayList<Student>();
for(Student s : students) {
    if(s.getScore() > 90.0) {
        studentsScore.add(s);
    }
}
Collections.sort(studentsScore, new Comparator<Student>() {
    public int compare(Student s1, Student s2) {
        return Double.compare(
           s1.getScore(), s2.getScore());
    }
});
```

Muito verboso quando comparamos com a implementação no Java 8 usando streams:

java Copiar Editar

```java
List<Student> studentsScore = students
        .stream()
        .filter(s -> s.getScore() >= 90.0)
        .sorted(Comparator.comparing(Student::getScore))
        .collect(Collectors.toList());
```

Não se preocupe se você não entender completamente o código; veremos o que ele significa mais adiante.

O que são streams?

Primeiro, **streams não são coleções**.

Uma definição simples é que **streams são ENVOLTÓRIOS** para coleções e arrays. Eles envolvem uma coleção EXISTENTE (ou outra fonte de dados) para suportar operações expressas com **lambdas**, ou seja, você especifica **o que** quer fazer, não **como** fazer.

Estas são as características de um stream:

- Streams funcionam perfeitamente com lambdas.
- Todas as operações de streams recebem interfaces funcionais como argumentos, permitindo simplificar o código com expressões lambda (e referências de método).
- **Streams não armazenam seus elementos.**
- Os elementos são armazenados numa coleção ou gerados dinamicamente. Eles apenas são carregados da fonte através de um pipeline de operações.
- **Streams são imutáveis.**
- Streams não alteram sua fonte subjacente de elementos. Se, por exemplo, um elemento for removido do stream, um novo stream é criado com esse elemento removido.
- **Streams não são reutilizáveis.**
- Eles podem ser percorridos apenas uma vez. Após uma operação terminal ser executada, é preciso criar outro stream a partir da fonte para processá-lo novamente.
- **Streams não oferecem acesso indexado** aos seus elementos.
- Streams não são coleções ou arrays. O máximo que se pode fazer é obter o primeiro elemento.
- **Streams são facilmente paralelizáveis.**
- Com a chamada de um método (e seguindo certas regras), é possível fazer com que um stream execute suas operações de forma concorrente, sem escrever nenhum código de multithreading.
- **Operações de stream são preguiçosas (lazy) quando possível.**
- Streams adiam a execução de suas operações até que os resultados sejam necessários ou até que se saiba quanta informação será necessária.

Para configurar esse pipeline, você:

1. Cria o stream.
2. Aplica zero ou mais operações intermediárias para transformar o stream inicial em novos streams.
3. Aplica uma operação terminal para gerar um resultado ou um "efeito colateral".

Vamos explicar essas etapas e, por fim, falaremos mais sobre preguiça (lazy). Nos capítulos subsequentes, veremos todas as operações suportadas por streams.

Criando streams

Um stream é representado pela interface java.util.stream.Stream<T>. Ela funciona apenas com objetos.

Há também especializações para trabalhar com tipos primitivos, como IntStream, LongStream e DoubleStream.

Existem muitas formas de criar um stream. Vamos começar com as três mais populares.

A primeira é criando um stream a partir de uma implementação de java.util.Collection usando o método stream():

```java
List<String> words = Arrays.asList(new String[]{"hello", "hola", "hallo", "ciao"});
Stream<String> stream = words.stream();
```

A segunda é criando um stream a partir de valores individuais:

```java
Stream<String> stream = Stream.of("hello","hola", "hallo", "ciao");
```

A terceira é criando um stream a partir de um array:

```java
String[] words = {"hello", "hola", "hallo", "ciao"};
Stream<String> stream = Stream.of(words);
```

No entanto, você precisa ter cuidado com este último método ao trabalhar com primitivos.

Veja por quê. Suponha um array de int:

```java
int[] nums = {1, 2, 3, 4, 5};
```

Quando criamos um stream a partir desse array assim:

```java
Stream.of(nums)
```

Não estamos criando um Stream<Integer>, mas um Stream<int[]>. Isso significa que, em vez de termos um stream com cinco elementos, temos um stream com **um único elemento**:

```java
System.out.println(Stream.of(nums).count()); // Imprime 1!
```

A razão está nas assinaturas do método of:

```java
// retorna um stream com um único elemento
static <T> Stream<T> of(T t)
// retorna um stream cujos elementos são os valores especificados
static <T> Stream<T> of(T... values)
```

Como int não é um objeto, mas int[] é, o método escolhido para criar o stream é o primeiro (Stream.of(T t)), não o com varargs, então um Stream<int[]> é criado. Mas como apenas um array é passado, o resultado é um stream com **um único elemento**.

Para resolver isso, podemos forçar o Java a escolher a versão com varargs criando um array de objetos (Integer):

```java
Integer[] nums = {1, 2, 3, 4, 5};
// Imprime 5!
System.out.println(Stream.of(nums).count());
```

Ou usar uma **quarta forma** de criar um stream (que, na verdade, é usada dentro do Stream.of(T... values)):

```java
int[] nums = {1, 2, 3, 4, 5};
// Também imprime 5!
System.out.println(Arrays.stream(nums).count());
```

Ou usar a versão primitiva IntStream:

```java
int[] nums = {1, 2, 3, 4, 5};
// Também imprime 5!
System.out.println(IntStream.of(nums).count());
```

Lição aprendida: **Não use Stream<T>.of() ao trabalhar com primitivos.**

A seguir, outras maneiras de criar streams:

```java
static <T> Stream<T> generate(Supplier<T> s)
```

Esse método retorna um stream "infinito" onde cada elemento é gerado pelo Supplier fornecido. Ele geralmente é usado com o método:

```java
Stream<T> limit(long maxSize)
```

Que trunca o stream para que ele não tenha mais que maxSize elementos.
Exemplo:

```java
Stream<Double> s = Stream.generate(new Supplier<Double>() {
    public Double get() {
        return Math.random();
    }
}).limit(5);
```

Ou:

```java
Stream<Double> s = Stream.generate(() -> Math.random()).limit(5);
```

Ou simplesmente:

```java
Stream<Double> s = Stream.generate(Math::random).limit(5);
```

Isso gera um stream de cinco números double aleatórios.

Operações Intermediárias

Você pode facilmente identificar operações intermediárias; elas sempre retornam **um novo stream**. Isso permite que as operações sejam encadeadas.

Exemplo:

```java
Stream<String> s = Stream.of("m", "k", "c", "t")
    .sorted()
    .limit(3);
```

Uma característica importante das operações intermediárias é que elas **não processam os elementos** até que uma **operação terminal** seja invocada, ou seja, **são preguiçosas (lazy)**.

As operações intermediárias se dividem ainda em **stateless** (sem estado) e **stateful** (com estado).

- Operações *stateless* não retêm estado dos elementos anteriores ao processar um novo elemento, então cada um pode ser processado de forma independente.
- Operações *stateful*, como distinct e sorted, podem incorporar estado de elementos já vistos ao processar novos elementos.

A tabela a seguir resume os métodos da interface Stream que representam operações intermediárias:

Método	**Tipo**	**Descrição**
Stream<T> distinct()	Stateful	Retorna um stream consistindo dos elementos distintos.
Stream<T> filter(Predicate<? super T> predicate)	Stateless	Retorna um stream de elementos que correspondem ao predicado fornecido.
<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)	Stateless	Retorna um stream com o conteúdo produzido aplicando a função fornecida a cada elemento. Há versões para int, long e double.
Stream<T> limit(long maxSize)	Stateful	Retorna um stream truncado para não ter mais que maxSize elementos.
<R> Stream<R> map(Function<? super T,? extends R> mapper)	Stateless	Retorna um stream consistindo dos resultados da aplicação da função dada aos elementos deste stream. Há versões para int, long e double.
Stream<T> peek(Consumer<? super T> action)	Stateless	Retorna um stream com os elementos deste stream, executando a ação fornecida em cada elemento.
Stream<T> skip(long n)	Stateful	Retorna um stream com os elementos restantes após descartar os n primeiros elementos.
Stream<T> sorted()	Stateful	Retorna um stream ordenado conforme a ordem natural dos seus elementos.
Stream<T> sorted(Comparator<? super T> comparator)	Stateful	Retorna um stream ordenado conforme o Comparator fornecido.

Método	Tipo	Descrição
Stream<T> parallel()	N/A	Retorna um stream equivalente que é paralelo.
Stream<T> sequential()	N/A	Retorna um stream equivalente que é sequencial.
Stream<T> unordered()	N/A	Retorna um stream equivalente que é desordenado.

Operações Terminais

Você também pode identificar facilmente as operações terminais: elas sempre retornam **algo diferente de um stream**.

Depois que a operação terminal é executada, o pipeline do stream é consumido e **não pode mais ser usado**. Exemplo:

```java
int[] digits = {0, 1, 2, 3, 4 , 5, 6, 7, 8, 9};
IntStream s = IntStream.of(digits);
long n = s.count();
System.out.println(s.findFirst()); // Uma exceção é lançada
```

Se você precisar percorrer o mesmo stream novamente, **precisa voltar à fonte de dados para obter um novo**. Exemplo:

```java
int[] digits = {0, 1, 2, 3, 4 , 5, 6, 7, 8, 9};
long n = IntStream.of(digits).count();
System.out.println(IntStream.of(digits).findFirst()); // OK
```

A tabela a seguir resume os métodos da interface Stream que representam operações terminais:

Método	Descrição
boolean allMatch(Predicate<? super T> predicate)	Retorna se **todos** os elementos do stream correspondem ao predicado fornecido.
boolean anyMatch(Predicate<? super T> predicate)	Retorna se **algum** elemento do stream corresponde ao predicado fornecido.
boolean noneMatch(Predicate<? super T> predicate)	Retorna se **nenhum** elemento do stream corresponde ao predicado fornecido.
Optional<T> findAny()	Retorna um Optional descrevendo **algum** elemento do stream.
Optional<T> findFirst()	Retorna um Optional descrevendo o **primeiro** elemento do stream.
<R,A> R collect(Collector<? super T,A,R> collector)	Executa uma operação de redução mutável nos elementos do stream usando um Collector.
long count()	Retorna a contagem de elementos no stream.

Método	Descrição
void forEach(Consumer<? super T> action)	Executa uma ação para cada elemento do stream.
void forEachOrdered(Consumer<? super T> action)	Executa uma ação para cada elemento, na ordem de encontro do stream (se definida).
Optional<T> max(Comparator<? super T> comparator)	Retorna o elemento **máximo** do stream de acordo com o Comparator fornecido.
Optional<T> min(Comparator<? super T> comparator)	Retorna o elemento **mínimo** do stream de acordo com o Comparator fornecido.
T reduce(T identity, BinaryOperator<T> accumulator)	Realiza uma redução nos elementos usando o valor identidade e uma função associativa de acumulação, retornando o valor reduzido.
Object[] toArray()	Retorna um array contendo os elementos do stream.
<A> A[] toArray(IntFunction<A[]> generator)	Retorna um array contendo os elementos do stream, usando a função geradora fornecida para alocar o array.
Iterator<T> iterator()	Retorna um iterador para os elementos do stream.
Spliterator<T> spliterator()	Retorna um *spliterator* para os elementos do stream.

Operações Preguiçosas (Lazy)

As **operações intermediárias** são **adiadas** até que uma **operação terminal** seja invocada.
O motivo é que as operações intermediárias geralmente podem ser **mescladas ou otimizadas** por uma operação terminal.

Vamos pegar, por exemplo, este pipeline de stream:

Aqui está o que ele faz:

- Gera um stream de strings,
- Depois converte o stream para um stream de int (representando o comprimento de cada string),
- Em seguida, filtra os comprimentos maiores que três,
- Depois, pega os dois primeiros elementos da sequência,
- E, finalmente, imprime esses dois elementos.

Você poderia pensar que a operação map é aplicada a todos os sete elementos, depois a operação filter também em todos, e então limit pega os dois primeiros, e por fim eles são impressos.

Mas não é assim que funciona.
Se modificarmos as expressões lambda de map e filter para exibir uma mensagem:

```java
Stream.of("sun", "pool", "beach", "kid", "island", "sea", "sand")
    .map(str -> {
        System.out.println("Mapping: " + str);
        return str.length();
    })
    .filter(i -> {
        System.out.println("Filtering: " + i);
        return i > 3;
    })
    .limit(2)
    .forEach(System.out::println);
```

A ordem de avaliação será revelada:

```makefile
Mapping: sun
Filtering: 3
Mapping: pool
Filtering: 4
4
Mapping: beach
Filtering: 5
5
```

A partir desse exemplo, podemos ver que o stream **não aplicou todas as operações no pipeline a todos os elementos**, mas apenas **até encontrar os elementos necessários para retornar um resultado** (devido à operação limit(2)).
Isso é chamado de **short-circuiting** (curto-circuito).

Operações de Curto-Circuito

As operações de curto-circuito fazem com que as operações intermediárias sejam processadas **apenas até que um resultado possa ser produzido**.

Dessa forma, por causa das operações preguiçosas e de curto-circuito, **streams não executam todas as operações em todos os seus elementos**, mas sim **até o ponto em que um resultado pode ser deduzido ou gerado**.

Você pode ver o curto-circuito como uma subcategoria. Há apenas uma operação intermediária de curto-circuito, enquanto o restante são terminais:

INTERMEDIÁRIA

- Stream<T> limit(long maxSize)

(*Porque não é necessário processar todos os elementos do stream para criar um stream de tamanho fixo*)

TERMINAIS

- boolean anyMatch(Predicate<? super T> predicate)
- boolean allMatch(Predicate<? super T> predicate)
- boolean noneMatch(Predicate<? super T> predicate)
- Optional<T> findFirst()

- Optional<T> findAny()

(*Porque assim que um elemento correspondente for encontrado, não há necessidade de continuar processando o stream*)

Pontos-Chave

- **Streams** podem ser definidos como **envoltórios (wrappers)** para coleções e arrays.
  Eles envolvem uma coleção existente (ou outra fonte de dados) para suportar operações expressas com **lambdas**, ou seja, você especifica **o que** quer fazer, não **como** fazer.

As características de um stream:

- Streams funcionam perfeitamente com lambdas.
- Streams não armazenam seus elementos.
- Streams são imutáveis.
- Streams não são reutilizáveis.
- Streams não suportam acesso indexado aos seus elementos.
- Streams são facilmente paralelizáveis.
- As operações de streams são preguiçosas (lazy), sempre que possível.

As operações de stream podem ser encadeadas para formar um pipeline.
Para configurar esse pipeline, você:

1. Cria o stream.
2. Aplica zero ou mais operações intermediárias para transformar o stream inicial em novos streams.
3. Aplica uma operação terminal para gerar um resultado ou um “efeito colateral”.

Existem várias formas de criar um stream. As mais populares são:

A partir de uma coleção existente:

```java
List<String> words = Arrays.asList(new String[]{"hello", "hola", "hallo", "ciao"});
Stream<String> s1 = words.stream();
```

A partir de elementos individuais:

```java
Stream<String> s2 = Stream.of("hello", "hola", "hallo", "ciao");
```

A partir de um array:

```java
String[] words = {"hello", "hola", "hallo", "ciao"};
Stream<String> s3 = Stream.of(words);
```

Não use Stream<T>.of() ao trabalhar com primitivos.
Em vez disso, use Arrays.stream ou as versões primitivas de Stream:

```java
int[] nums = {1, 2, 3, 4, 5};
IntStream s1 = Arrays.stream(nums);
IntStream s2 = IntStream.of(nums);
```

- **Operações intermediárias**, como map ou filter, sempre retornam um novo stream, e são divididas entre **stateless** e **stateful**.
  Além disso, são **preguiçosas**, ou seja, são adiadas até que uma operação terminal seja invocada.
- **Operações terminais**, como count ou forEach, sempre retornam algo **diferente de um stream**.
- **Operações de curto-circuito** fazem com que operações intermediárias sejam processadas apenas até que um resultado possa ser produzido.

Assim, por conta da preguiça e do curto-circuito, os streams não executam todas as operações sobre todos os seus elementos, mas sim até o ponto onde um resultado possa ser deduzido ou gerado.

Autoavaliação

1. Dado:

```java
public class Question_12_1 {
    public static void main(String[] args) {
        IntStream.range(1, 10)
            .filter(i -> {
                System.out.print("1");
                return i % 2 == 0;
            })
            .filter(i -> {
                System.out.print("0");
                return i > 3;
            })
            .limit(1)
            .forEach(i -> {
                System.out.print(i);
            });
    }
}
```

Qual é o resultado?
A. 101010104
B. 1111111110000000004
C. 11041106
D. 1101104
E. Uma exceção é lançada

---

2. Quais das seguintes são operações intermediárias?

A. limit
B. peek
C. anyMatch
D. skip

---

3. Quais das seguintes são operações terminais?

A. sorted
B. flatMap
C. max
D. distinct

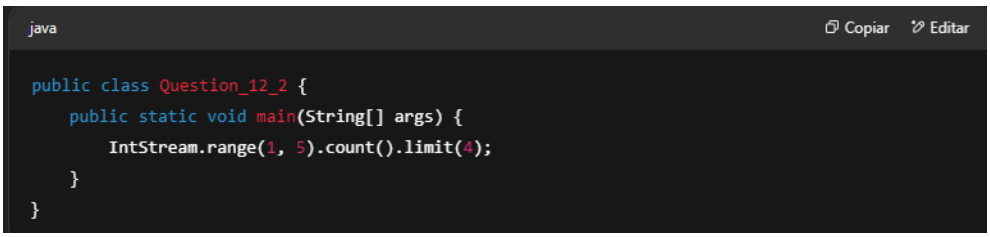
---

4. Quais das seguintes são operações de curto-circuito?

A. reduce
B. parallel
C. findNone
D. findFirst

---

5. Dado:



Qual é o resultado?
A. 5
B. 4
C. 1
D. Erro de compilação
E. Uma exceção é lançada